PGR112 – Step 9: Interface, polymorphism and casting

Object Oriented Programming

Bogdan Marculescu / Bogdan.Marculescu@kristiania.no

Agenda

- Interface
- Polymorphism
- Casting

Quick recap of abstract classes

- We discussed abstraction, abstract methods and classes.
- We used abstraction to describe **what** a class does, but not **how.**
- We used it to hide implementation details.
- The degree of abstraction may vary within a class:
  - No abstract methods
  - Some abstract methods
  - Only abstract methods

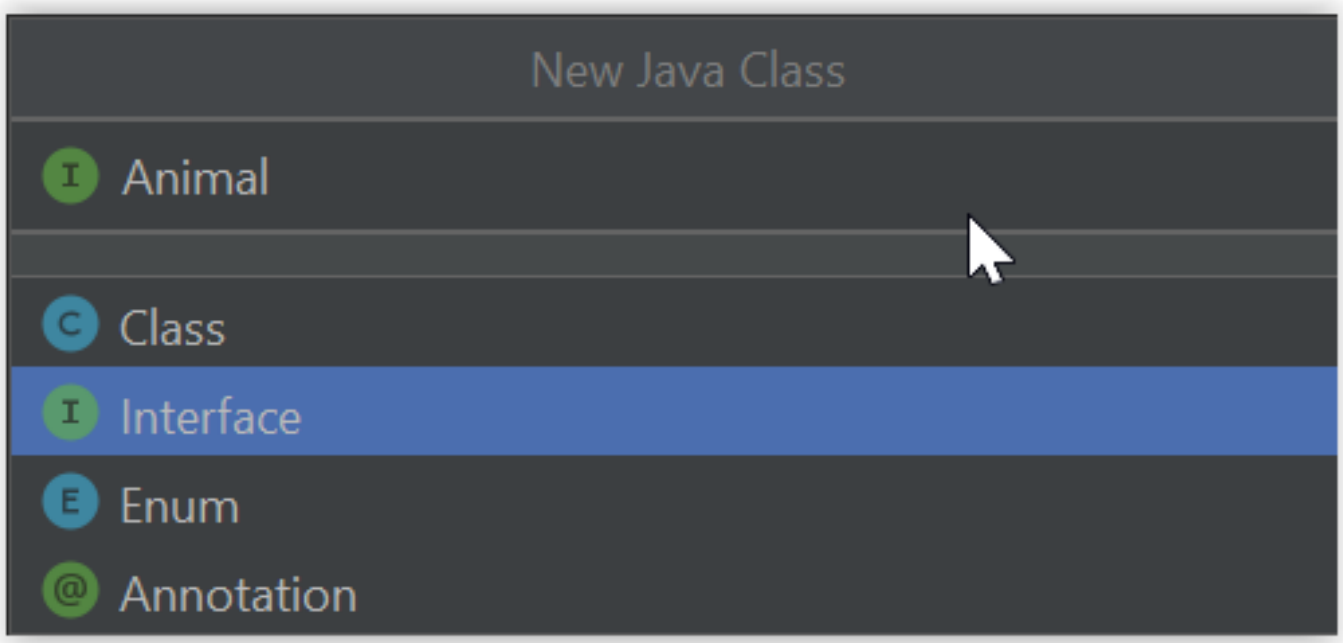
- In interface we can say that we have 100% abstraction (but that is not entirely true - more on that later)

Interface

- An interface can be described as a kind of contract: What the class commits to offering in terms of methods.
- Remember, a class can only inherit from one class. But a class can implement multiple interfaces.
- Let's take a well-known example from w3schools.

Creating an interface (IntelliJ)

- The same way we create classes (Rclick folder you want to put the interface in -> new -> Java class)
- Select the name of the interface and select Interface:

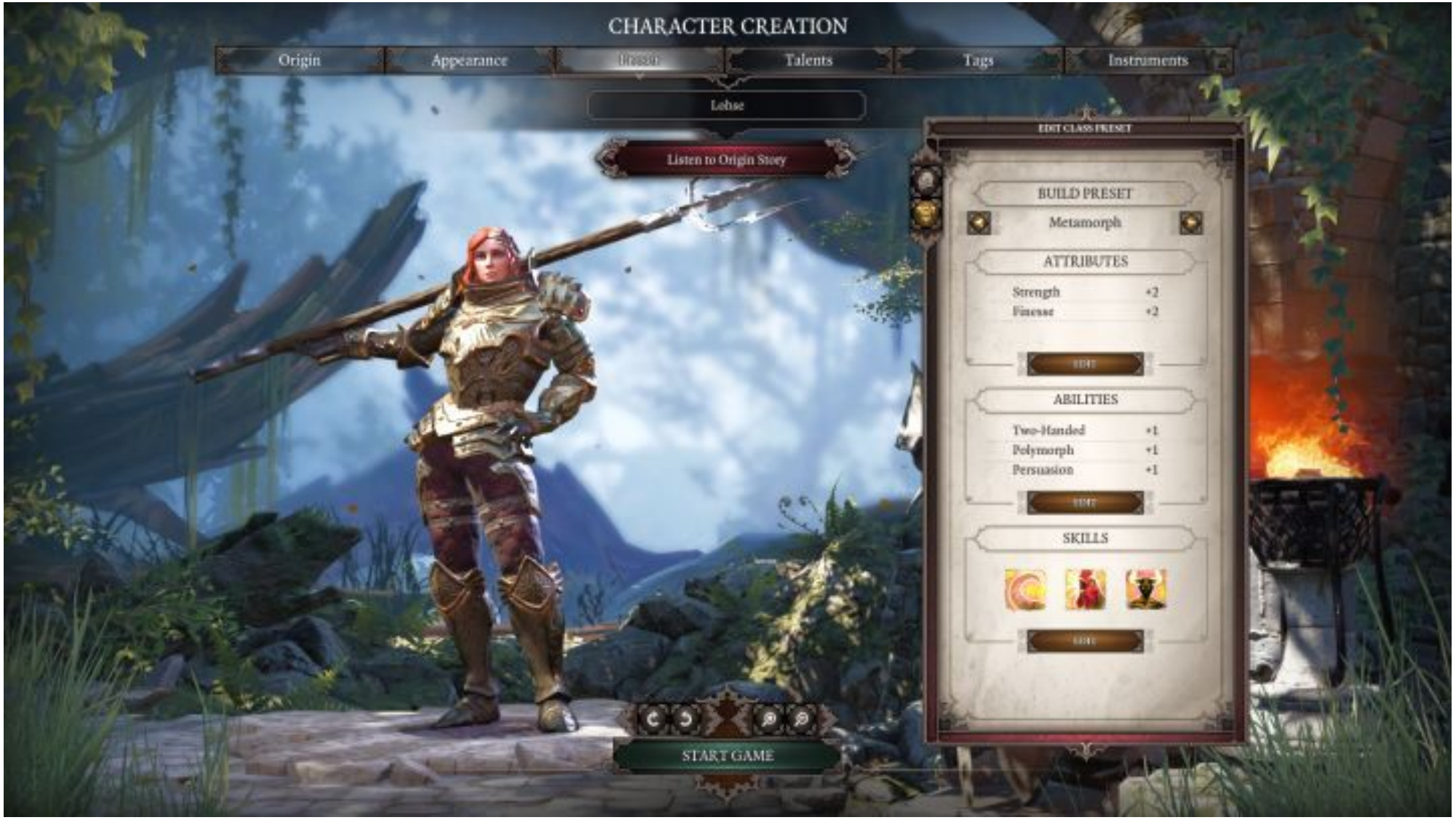


Some rules for interfaces

- We can not create objects by interface (same as for abstract classes).
- The methods do not have a method body (same as for abstract methods).
- A class that implements an interface MUST implement all the methods in the interface (or be abstract). Same as we have seen with abstract classes.
- We do not need to specify that the methods are public and abstract (because they always are)
- They may have attributes, but it's unusual (and not something we get into in this topic)

Polymorphism

Gamepressure.com



Polymorhism

- Poly = many/several
- Morph = form
- In Java, we have two types of polymorphism:
    1. Compile time polymorphism
    2. Runtime polymorphism

Compile time polymorphism

- Also called static polymorphism
- Achieved by function overloading or operator overloading.

Java does not support this.

Example of compile time polymorphism

- Two methods have the same name
- Takes different types of parameters (int, double)
- Which method is called is decided "compile time"

```java
// Java program to demonstrate
// compile-time polymorphism
public class GFG {

    // First addition function
    public static int add(int a, int b)
    {
        return a + b;
    }

    // Second addition function
    public static double add(
        double a, double b)
    {
        return a + b;
    }

    // Driver code
    public static void main(String args[])
    {
        // Here, the first addition
        // function is called
        System.out.println(add(2, 3));

        // Here, the second addition
        // function is called
        System.out.println(add(2.0, 3.0));
    }
}
```

geeksforgeeks

Runtime polymorphism

- Also called Dynamic Method Dispatch
- Which method is called is decided during runtime.
- We achieve this by method overriding

Example of runtime polymorphism

- The GFG class is a subclass of the Test class.
- In GFG we override the method «method»
- In the main method we have an object reference of type Test.
- But this refers to an object of type GFG.
- When the method is called, the subclass version is run.
- This is decided in runtime.

```java
// Java program to demonstrate
// runtime polymorphism

// Implementing a class
class Test {

    // Implementing a method
    public void method()
    {
        System.out.println("Method 1");
    }
}

// Defining a child class
public class GFG extends Test {

    // Overriding the parent method
    public void method()
    {
        System.out.println("Method 2");
    }

    // Driver code
    public static void main(String args[])
    {
        Test test = new GFG();

        test.method();
    }
}
```

geeksforgeeks

Upcasting

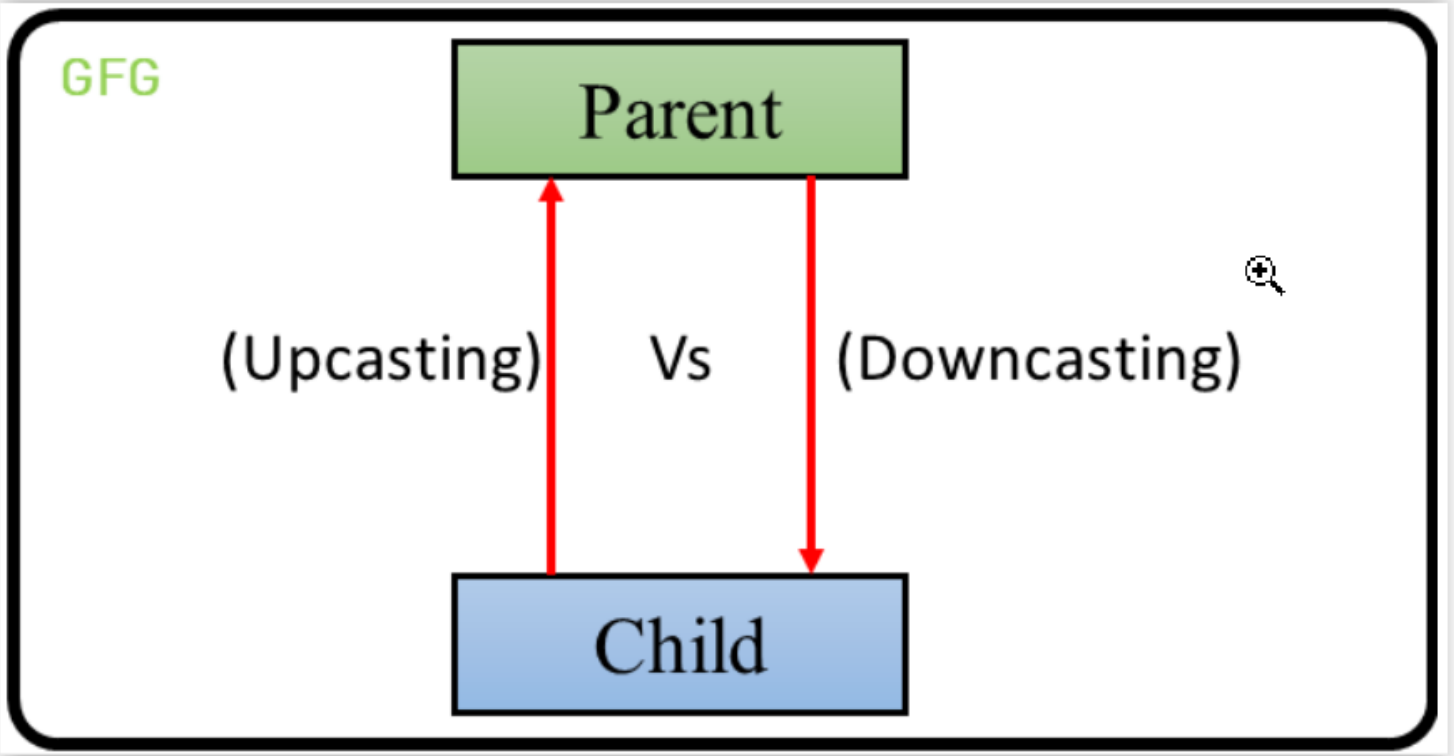
- When the reference to object C is of parent type P.
- From the previous slide: Test t = new GFG ();
- We do not have access to fields or methods that are unique to the subclass.
- But if we call a method that the subclass overrides, then it is the subclass version that runs (as we saw in the previous example - runtime polymorphism).
- Let's take another demo example...

We have already used runtime polymorphism in previous steps

- Remember?

Two types of casting

- Upcasting (which we've seen)
- Downcasting



geeksforgeeks

Downcasting

- Let's imagine that we have a reference (p) to a parent class Parent. But we know that the object referred to is of a subclass Child.
- We can downcast: Child c = (Child) p;
- If it went well, then we now have access to all fields and methods in the subclass.

Downcasting

- A little "scarier" than upcasting.
- For the children's class can only have one parent. Therefore, upcasting is safe.
- But a parent can have many children (subclasses).
- If we try to downcast, but the object is not actually of the correct type, what happens then? ClassCastException!
- Let's see it with our own eyes…

instanceof

- We can check if an object is of a certain type using the instanceof operator.
- We can therefore protect ourselves against ClassCastException.
- For example:

```
if (p instanceof Child) {
        Child c = (Child)p;
}
```

Let’s try this out a bit…

Interface, cont.

- I mentioned earlier that the interface is 100% abstract, but that this is not entirely the case…
- A few years ago, default methods were allowed in the interface.
- The starting point is that if we change an interface, then already existing classes that implement the interface will have problems.
- Imagine that you have some classes that implement an interface from the Java library. In a newer version of Java, the interface is expanded with an additional method. Your classes no longer compile
- Therefore, you can enter default methods (implementations) in the interface that ensure that this does not happen.
- You can read more here.

A little more info on interfaces

- An interface can inherit from another interface (in several levels). Uses extends that we know from before.
- We also have something called "tagging interfaces": interface without methods. Two purposes:
  1. Create a unifying parenting interface.
  2. Tag a class (hence the origin of "tagging interfaces")
- Some examples of tagging interfaces (interface without methods):
  - java.util.EventListener
  - java.io.Serializable

Why use interfaces?

- We have seen that it is a form of abstraction.
- We can achieve a form of multiple inheritance (which we can not get with the use of extends)
- Another very good reason is **loose coupling**.
- Let's take a well-known example…

Conclusion

- Goals for this step:
  - I can use the interface, and I understand what that means.
  - I understand what polymorhism is.
  - I know how to use casting.

Good luck with the exercises.